```
        {
            for(i=0; i<m_numPixels; i++)
            {
                if(show_progress && i%100==0) load_progress.SetPos((99*i)/m_numPixels);
                p=GetPixel(i); if(p>=color_map_size) p=color_map_size-1;
                SetPixel(i,color_map[p]);
            }
        }
        return true;
    }
}

bool Image::SetPalette(int px, int py, int pxmax, int pymax)    // set palette with respect to the mouse posit
ion
{
    int offset=(((long)m_maxPixelValue/3)*(2*px-pxmax))/pxmax;
    double x=(2.0*py)/pymax;
    x=1+2*(x-1)*(x-1)*(x-1);
    return m_pPal->SetPalette(offset,x);
}



/*****************************************************************************
*
*   Plot DIB pixels
*
*****************************************************************************/
bool Image::DisplayDIB(CDC *pDC, CRect crectScreenArea, CRect crectImageArea,
                       CPoint pScroll, bool optimize)
{
    UINT palUsage;

    if(m_DisplayFailed) return false;
    if(m_pPal->p_active && m_pPal->LoadPalette(pDC,m_RGB))
    {
        palUsage=DIB_PAL_COLORS;
    }
    else palUsage=DIB_RGB_COLORS;

    /* Optimize drawing regions */
    if(optimize)
    {
        CRect rcClip, rcDraw, rcDIB;

        pDC->GetClipBox(rcClip);
        rcClip.NormalizeRect();
        if(rcClip.IsRectEmpty())    return true;
        rcClip.InflateRect(2,2);
        rcDraw.IntersectRect(rcClip,crectScreenArea);
        rcDraw.NormalizeRect();
        if(rcDraw.IsRectEmpty())    return true;

        rcDIB=m_smS.Screen_to_Image(rcDraw);
        if(rcDIB.IsRectEmpty())     return true;

        crectScreenArea.CopyRect(rcDraw);
        if(theApp.app_Metheus)  crectScreenArea.OffsetRect(-pScroll);
        crectImageArea.CopyRect(rcDIB);
    }

    /* Correct image area */
    if(crectImageArea.left<0)   crectImageArea.OffsetRect(-crectImageArea.left,0);
    else if (crectImageArea.right>m_Width)
        crectImageArea.OffsetRect(m_Width-crectImageArea.right,0);
    if(crectImageArea.top <0)   crectImageArea.OffsetRect(0,-crectImageArea.top);
    else if (crectImageArea.bottom>m_Height)
        crectImageArea.OffsetRect(0,m_Height-crectImageArea.bottom);

    /* Set screen (destination) and bitmap (source) areas */
    long xDest=crectScreenArea.TopLeft().x; long yDest=crectScreenArea.TopLeft().y;
    long wDest=crectScreenArea.Width();     long hDest=crectScreenArea.Height();
    long xBmp=crectImageArea.TopLeft().x;
    long yBmp;
    if(theApp.app_Metheus)  yBmp=crectImageArea.TopLeft().y;
    else
    {
        yBmp=this->GetHeight()-crectImageArea.TopLeft().y-crectImageArea.Height();
    }
```

```
    long wBmp=crectImageArea.Width();       long hBmp=crectImageArea.Height();
    switch(m_Flip)
    {
    case 1: // horizontal
        xDest += wDest-1;   xBmp=this->GetWidth()-xBmp-wBmp;
        wDest = -wDest;     break;
    case 2: // vertical
        yDest += hDest-1;   yBmp=this->GetHeight()-yBmp-hBmp;
        hDest = -hDest;     break;
    case 3: // 180
        xDest += wDest-1;   xBmp=this->GetWidth()-xBmp-wBmp;
        wDest = -wDest;     yDest += hDest-1;
        yBmp=this->GetHeight()-yBmp-hBmp;   hDest = -hDest; break;
    }

    /* Display as DIB */
    if(wDest==0 || hDest==0 || wBmp ==0 || hBmp==0) return true;
    if(!theApp.app_Metheus)
    {
        m_DisplayFailed= (StretchDIBits( pDC->GetSafeHdc(),
                        xDest, yDest,   wDest,  hDest,  xBmp,   yBmp,   wBmp,   hBmp,
                        m_Pixels, m_bmpInfo, palUsage, SRCCOPY)<=0);
    }
    else    // Metheus
    {
        m_DisplayFailed=true;
        if(m_UpdateBitmap)
        {
            if(m_Bitmap)
                MetheusDeleteCompatibleBitmap(pDC->GetSafeHdc(),m_Bitmap);
            m_Bitmap = MetheusCreateCompatibleBitmap(pDC->GetSafeHdc(),GetWidth(), GetHeight());
            if(MetheusLoadImageFromData(pDC->GetSafeHdc(), m_Bitmap,
                theApp.app_DynamicPaletteStart,m_Pixels, GetWidth(), GetHeight(),
                GetWidth()*m_Bytes_per_Pixel,8*m_Bytes_per_Pixel,
                0, 0,  GetWidth(), GetHeight(), 0,0)==FALSE)
            {
                AfxMessageBox("Cannot create image bitmap", MB_OK | MB_ICONEXCLAMATION);
                return  false;
            }
        }
            m_DisplayFailed=
            (MetheusStretchBltImage(pDC->GetSafeHdc(), NULL,
                                xDest, yDest, wDest, hDest,
                                m_Bitmap,
                                xBmp, yBmp,wBmp,hBmp,SRCCOPY)==FALSE);

    }
    if(m_DisplayFailed) // Display error
    {
        DWORD ecode=GetLastError();
        CString estr;
        estr.Format(" GDI error code: %ld\n Please reload the image", ecode);
        AfxMessageBox(estr, MB_OK | MB_ICONEXCLAMATION);
    }
    m_UpdateBitmap=false;
    return (!m_DisplayFailed);
}
bool Image::DisplayDIB(CDC *pDC)
{
    return DisplayDIB(pDC,m_smS.crScreen,m_smS.crImage, CPoint(0,0), true);
}
bool Image::DisplayDIB(CDC *pDC, CPoint pScroll)
{
    return DisplayDIB(pDC,m_smS.crScreen,m_smS.crImage, pScroll, true);
}


/*****************************************************************************
*
*   Get image width
*
*****************************************************************************/
int Image::GetWidth()
{
    return m_Width;
}


/*****************************************************************************
*
```

```
    s1 += t*1.09861229; s2 += t;
    // d=sqrt(10)
    p=GetLuminance(x-1,y-3);    pmax=pmin=p;
    p=GetLuminance(x-1,y+3);    if(p>pmax) pmax=p; else if(p<pmin) pmin=p;
    p=GetLuminance(x+1,y-3);    if(p>pmax) pmax=p; else if(p<pmin) pmin=p;
    p=GetLuminance(x+1,y+3);    if(p>pmax) pmax=p; else if(p<pmin) pmin=p;
    p=GetLuminance(x-3,y-1);    if(p>pmax) pmax=p; else if(p<pmin) pmin=p;
    p=GetLuminance(x-3,y+1);    if(p>pmax) pmax=p; else if(p<pmin) pmin=p;
    p=GetLuminance(x+3,y-1);    if(p>pmax) pmax=p; else if(p<pmin) pmin=p;
    p=GetLuminance(x+3,y+1);    if(p>pmax) pmax=p; else if(p<pmin) pmin=p;
    t=log(max(pmax-pmin,1));
    s1 += t*1.15129255; s2 += t;
    s1 *= 0.14285714;   s2 = s2*0.10477684;

    //Find fractal dimension as slope of the Hurst line
    //t=max(6.386491206*(s1-s2),0.0); // fractal dimension
    t=max(600*m_TR_scale*(s1-s2),0.0); // fractal dimension
    return( (long)(t) );
}

/*****************************************************************************
*
*   Apply a pixel neighborhood function (with code mask_type) to the image
*
*****************************************************************************/
bool Image::TR_PixelNeighboorhood(char mask_type, bool show_progress)
{
    int rad,i,j,j1,top,bot;
    long p, pmax, pcount;
    double pavg;

    // Set pointer to the masking function
    long (Image::*maskF)(const int x, const int y) = 0;
    switch(mask_type)
    {
    case 'g': maskF=TR_Gauss;  rad=1;   break; // gaussian denoising
    case 'a': maskF=TR_Smooth; rad=1;   break; // average smoothing
    case 's': maskF=TR_Sharp;  rad=1;   break; // sharpening
    case 'e': maskF=TR_Sobel;  rad=1;   break; // edge detector
    case 'f': maskF=TR_Fractal;  rad=3; break; // edge detector
    default: return false; // invalid mask type
    }

    // Create temporary buffer
    int w=this->GetWidth();
    int h=this->GetHeight();
    long *(*buf)=new long*[rad+1];
    if(!buf)
    {
        AfxMessageBox("Low memory, cannot transform");
        return false;
    }
    for(i=0; i<=rad; i++)
    {
        buf[i] = new long[w];
        if(buf[i]==0)
        {
            AfxMessageBox("Low memory, cannot transform");
            for(j=0; j<i; j++)  {   if(buf[j]) delete [] buf[j];    }
            delete [] buf;
            return false;
        } // out of memory
    }


    // Display progress control in the main frame status bar
    CProgressCtrl load_progress;
    if(show_progress)
    {
        CreateProgressControl(load_progress,"Transforming ...");
        load_progress.SetPos(5);
    }

    // Estimate max transformed pixel value
    m_TR_scale=1.0;
    p=pmax=(this->*maskF)(rad+1,rad+1);
    pcount=0;   pavg=0.0;
    int dw = w>>4;  if(dw<1) dw=1;
    int dh = h>>4;  if(dh<1) dh=1;
    for(i=rad+1; i<w-rad; i += dw)
    {
```

```
        for(j=rad+1; j<h-rad; j     ih)
        {
            p=(this->*maskF)(i,j);
            pcount ++;
            pavg += p;
            if(p>pmax)  pmax=p;
        }
    }
    if(pmax>0 && pcount>1)
    {
        pavg /= pcount;
        pmax=min(pmax,(long)(2*pavg));
        m_TR_scale=0.6*(double)(m_maxPixelValue)/pmax;
    }
    if(show_progress) load_progress.SetPos(10);

    // Apply (2*rad-1)*(2*rad-1) masking operator
    bot=-1;
    top=rad-1;
    for(j=rad; j<=h; j++)
    {
        if(show_progress && j%50==0) load_progress.SetPos(10+(90*j)/h);
        jl=j-rad-1;
        if(jl>=rad)
        {
            for(i=rad; i<w-rad; i++) SetPixel(i,jl,buf[bot][i]);
        }
        bot = (bot+1)%(rad+1);
        top = (top+1)%(rad+1);
        if(j<h-rad)
        {
            for(i=rad; i<w-rad; i++) buf[top][i]=(this->*maskF)(i,j);
        }
    }

    // Clean up
    for(j=0; j<=rad; j++)   {   if(buf[j]) delete [] buf[j];    }
    delete [] buf;
    Beep(500,50);
    return true;
}
```

```
/*****************************************************************************
*
*   Inverts the image
*
*****************************************************************************/
bool Image::TR_Negate()
{
    if(m_pPal->p_active) m_pPal->Negate();
    else
    {
        // Display progress control in the main frame status bar
        CProgressCtrl load_progress;
        CreateProgressControl(load_progress, "Changing to negative image ...");
        // Invert pixels values
        if(m_RGB)
        {
            BYTE r, g, b;
            long p;
            for(long i=0; i<m_numPixels; i++)
            {
                if(i%1000==0) load_progress.SetPos((99*i)/m_numPixels);
                p=GetPixel(i);
                r=m_maxPixelValue-GetRValue(p);
                g=m_maxPixelValue-GetGValue(p);
                b=m_maxPixelValue-GetBValue(p);
                SetPixel(i,RGB(r,g,b));
            }
        }
        else
        {
```

```
            for(long i=0; i<m_num    els; i++)
            {
                if(i%1000==0) load_progress.SetPos((99*i)/m_numPixels);
                SetPixel(i,m_maxPixelValue-GetPixel(i));
            }
        }
    }
    Beep(500,100);
    return true;
}



/*****************************************************************************
*
*   Copy pixels from current to safe buffer (on true)
*   or vice versa (on false)
*
*****************************************************************************/
void Image::ResetPixels(bool current_to_safe)
{
    if(current_to_safe) // current -> safe
    {
        memcpy(m_Safe_Pixels,m_Pixels,m_numPixelBytes);
    }
    else    // safe -> current
    {
        memcpy(m_Pixels,m_Safe_Pixels,m_numPixelBytes);
        m_UpdateBitmap=true;
    }
}



/*****************************************************************************
*
*   Histogramm stretch from [amin,amax] to [bmin,bmax] color range.
*   R,G and B components are stretched together
*
*****************************************************************************/
bool Image::TR_HistStretch(int amin,int amax,int bmin,int bmax,
                           bool show_progress)
{
    long i, cmax, p;

    /* Create color map */
    if(m_pPal->p_active) cmax=m_pPal->p_Size;
    else Get_Pixel_minmax(i,cmax);
    cmax++;
    long* color_map=new long[cmax];
    if(!color_map)
    {
        AfxMessageBox("Low memory, cannot perform this transform");
        return false;
    } // out of memory

    /* Set color map parameters */
    if(amin<0) amin=0;
    if(bmin<0) bmin=0;
    if(amax>m_maxPixelValue) amax=m_maxPixelValue;
    if(bmax>m_maxPixelValue) bmax=m_maxPixelValue;
    if (amin>=amax || bmin>=bmax) return false; // invalid map
    if(amin==bmin && amax==bmax) return true; // no stretch needed

    /* Fill the color map */
    long da=amax-amin;
    long db=bmax-bmin;
    for(i=0; i<cmax; i++)
    {
        p=bmin+(db*(i-amin))/da;
        if(p<bmin) p=bmin;  else if (p>bmax) p=bmax;
        color_map[i]=p;
    }

    SetPalette(color_map, cmax, show_progress);

    delete [] color_map;
    return true;
}
```

```
/*****************************************************************************
*
*   Histogramm stretch from (percent)% median neighborhood
*   to the maximal [0,m_maxPixelValue] range
*
*****************************************************************************/
bool Image::TR_HistStretch(BYTE percent, bool show_progress)
{
   long i, amin, amax, amed, p;

    /* Validation */
    if (percent>=100) percent=99;
    if(percent<0) return true;

    /* Find pixel statistics */
    Get_Pixel_minmax(amin,amax);

    /* Initialize image histogram */
    long cmax=amax+1;
    int* hist=new int[cmax];
    if(!hist)
    {
        AfxMessageBox("Low memory, cannot perform this transform");
        return false;
    } // out of memory
    for(i=0; i<cmax; i++) hist[i]=0;

    /* Estimate image histogram */
    int hmax=20000;
    int di=__max(1,m_numPixels/10000);
    if(m_RGB) // Color image
    {
        for(i=0; i<m_numPixelBytes; i += di)
        {
            p=m_Pixels[i];
            if(hist[p]<hmax) hist[p] += di;
        }
    }
    else // Greyscale image
    {
        for(i=0; i<m_numPixels; i += di)
        {
            p=GetPixel(i);
            if(hist[p]<hmax) hist[p] += di;
        }
    }
    if(percent==0) return TR_HistStretch(amin,amax,0,
                          m_maxPixelValue,show_progress); // simple stretch

    /* Find histogram color average */
    double ptot=0, tot=0;
    for(i=0; i<cmax; i++)
    {
        ptot += ((double)i)*hist[i];
        tot += hist[i];
    }
    amed = (long)(0.5+ptot/tot);

    /* Find new intensity range to preserve */
    double keep_max=(100-percent)*tot/100;  // number of pixels to keep
    double keep=hist[amed];
    long bmin=amed; long bmax=amed;
    do // do at least once to guarantee bmin<bmax
    {
        if(bmin>amin)
        {
            bmin--;
            keep += hist[bmin];
        }
        if(bmax<amax)
        {
            bmax++;
            keep += hist[bmax];
        }
    } while (keep<=keep_max);
    delete [] hist;

    if( ((bmin!=amin)||(bmax!=amax)) && (bmin<bmax) )
```

```
        return TR_HistStretch(br    bmax,0,m_maxPixelValue,show_progress)
    else return false;
}


/*****************************************************************************
*
*   Histogramm equalization
*
*****************************************************************************/
bool Image::TR_HistEqualize(bool show_progress)
{
    long i, amin, amax, p;

    /* Find pixel statistics */
    Get_Pixel_minmax(amin,amax);
    Beep(300,100);

    /* Initialize image histogram, also used as color map */
    long cmax=amax+1;
    long* hist=new long[cmax];
    if(!hist)
    {
        AfxMessageBox("Low memory, cannot perform this transform");
        return false;
    } // out of memory
    for(i=0; i<cmax; i++) hist[i]=0;

    /* Compute image histogram */
    long hmax=2000000;
    int di=__max(1,m_numPixels/10000);
    if(m_RGB) // Color image
    {
        for(i=0; i<m_numPixelBytes; i += di)
        {
            p=m_Pixels[i];
            if(hist[p]<hmax) hist[p] += di;
        }
    }
    else // Greyscale image
    {
        for(i=0; i<m_numPixels; i += di)
        {
            p=GetPixel(i);
            if(hist[p]<hmax) hist[p] += di;
        }
    }

    /* Integrate the histogram */
    double htotal=0;
    for(i=0; i<cmax; i++)   htotal += hist[i];
    if(htotal<1) return false;  // did we have negative pixels or empty image ?
    if(hist[0]<htotal-1) { htotal -= hist[0]; hist[0]=0; }

    /* Fill the color map */
    double hcum=0;
    for(i=0; i<cmax; i++)
    {
        hcum += hist[i];    // update cumulative hist
        hist[i]=(long)((m_maxPixelValue*hcum)/htotal);
    }

    /* Remap the pixel data */
    SetPalette(hist, cmax, show_progress);

    /* Clean up */
    delete [] hist;
    return true;
}



/*****************************************************************************
*
*   Image Gamma correction
*
*****************************************************************************/
bool Image::TR_GammaCorrection(double gamma)
{
```

```
#if !defined(AFX_LUPA_H__24CE8B9   D8F_11D2_958F_000000000000__INCLUDED_
#define AFX_LUPA_H__24CE8B94_7D8F_11D2_958F_000000000000__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// Lupa.h : header file
//
#include "Image.h"

/////////////////////////////////////////////////////////////////////////////
// Lupa dialog

class Lupa : public CDialog
{
// Construction
public:
    Lupa(CWnd* pParent = NULL);
    ~Lupa();

// Dialog Data
    //{{AFX_DATA(Lupa)
    enum { IDD = IDD_MAGNIFY_DIALOG };
    private:
    BOOL    l_optimize;
    long    l_height;
    long    l_width;
    public:
    double  l_zoom;
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(Lupa)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
public:
    bool    l_active;
    CSize   l_scnSize;

    void    Initialize(CSize csScn, double scrn_zoom, double lupa_zoom);
    void    Reset_l_DC(CDC *pDC, CRect& scrolled_client);
    void    Move(CPoint& a, CPoint& rel, Image* pBmp, CDC* pDC);
    bool    Resize(CDC* pDC, CPoint center, CPoint vertex);
    bool    Resize(CDC *pDC);
    CString toString();
    inline CRect Lupa::SetImgRect(CPoint & cpI)
    {
        CRect r( CPoint(cpI.x-(l_imgSize.cx>>1), cpI.y-(l_imgSize.cy>>1)),  l_imgSize );
        return r;
    };
    inline CRect SetScrnRect(CPoint & cpS)
    {
        return CRect(CPoint(cpS.x-(this->l_scnSize.cx>>1), cpS.y-(this->l_scnSize.cy>>1)),
                     this->l_scnSize);
    };
protected:

    // Generated message map functions
    //{{AFX_MSG(Lupa)
    virtual BOOL OnInitDialog();
    afx_msg void OnChangeMagnifyWidthEdit();
    afx_msg void OnChangeMagnifyHeightEdit();
    afx_msg void OnChangeMagnifyZoomEdit();
    afx_msg void OnChangeMagnifyOptimize();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    bool    l_preactive;
    double  l_img_zoom;
    CSize   l_imgSize;
    CRect   l_scnRect, l_dragRect;
    HBITMAP l_DIB;
    BYTE*   l_Data;
    CDC*    l_DC;

    void    Draw(CDC *pDC);
```

```
    bool    Update2(CRect& a, CRe..&b);
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_LUPA_H__24CE8B94_7D8F_11D2_958F_000000000000__INCLUDED_)
```

```
// Lupa.cpp : implementation file
//

#include "stdafx.h"
#include "DCM.h"
#include "Lupa.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// Lupa dialog
Lupa::Lupa(CWnd* pParent /*=NULL*/)
    : CDialog(Lupa::IDD, pParent)
{
    //{{AFX_DATA_INIT(Lupa)
    l_height = 128*theApp.app_ResolutionScaleFactor;
    l_width = l_height;
    l_zoom = 2.0;
    l_optimize=false;
    //}}AFX_DATA_INIT
    l_active=false;
    l_preactive=false;
    l_DC=NULL;
    l_DIB=NULL;
    l_Data=NULL;
    this->Initialize(CSize(l_width,l_height), 1.0, l_zoom);
}

Lupa::~Lupa()
{
    if(l_DC && !theApp.app_Metheus) delete l_DC;
    if(l_Data)  delete [] l_Data;
    if(l_DIB)
    {
        MetheusDeleteCompatibleBitmap(l_DC->GetSafeHdc(),l_DIB);
    }
}


void Lupa::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(Lupa)
    DDX_Text(pDX, IDC_MAGNIFY_HEIGHT_EDIT, l_height);
    DDV_MinMaxLong(pDX, l_height, 0, 1000);
    DDX_Text(pDX, IDC_MAGNIFY_WIDTH_EDIT, l_width);
    DDV_MinMaxLong(pDX, l_width, 0, 1000);
    DDX_Text(pDX, IDC_MAGNIFY_ZOOM_EDIT, l_zoom);
    DDV_MinMaxDouble(pDX, l_zoom, 0., 10.);
    DDX_Check(pDX, IDC_MAGNIFY_OPTIMIZE, l_optimize);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(Lupa, CDialog)
    //{{AFX_MSG_MAP(Lupa)
    ON_EN_CHANGE(IDC_MAGNIFY_WIDTH_EDIT, OnChangeMagnifyWidthEdit)
    ON_EN_CHANGE(IDC_MAGNIFY_HEIGHT_EDIT, OnChangeMagnifyHeightEdit)
    ON_EN_CHANGE(IDC_MAGNIFY_ZOOM_EDIT, OnChangeMagnifyZoomEdit)
    ON_EN_CHANGE(IDC_MAGNIFY_OPTIMIZE, OnChangeMagnifyOptimize)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/*****************************************************************************
*   LUPA initialization
*
*   Input:
*   on-screen lupa size csScn, screen image zoom scrn_zoom, LUPA l_zoom  l_img_zoom
*
*****************************************************************************/
void Lupa::Initialize(CSize csScn, double scrn_zoom, double lupa_zoom)
{
    l_scnSize=CSize(csScn.cx,csScn.cy);
    l_zoom=lupa_zoom;
    l_img_zoom=scrn_zoom;
    double ivzf=1.0 / (scrn_zoom*lupa_zoom); //combined inversed zoom
```

```
    l_imgSize=CSize((long)(1+ivz.   _scnSize.cx),
                    (long)(1+ivzf-l_scnSize.cy));
    l_dragRect=CRect(0,0,0,0);
}

/*************************************************************************
*
*   Lupa message handlers
*
*************************************************************************/
BOOL Lupa::OnInitDialog()
{
    CDialog::OnInitDialog();
    l_width=l_scnSize.cx;
    l_height=l_scnSize.cy;
    UpdateData(FALSE);
    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}
void Lupa::OnChangeMagnifyWidthEdit()
{
    UpdateData(TRUE);
    if(l_width<10) l_width=10;
    l_scnSize.cx=l_width;
    Initialize(l_scnSize, l_img_zoom, l_zoom);
}
void Lupa::OnChangeMagnifyHeightEdit()
{
    UpdateData(TRUE);
    if(l_height<10) l_height=10;
    l_scnSize.cy=l_height;
    Initialize(l_scnSize, l_img_zoom, l_zoom);
}
void Lupa::OnChangeMagnifyZoomEdit()
{
    UpdateData(TRUE);
    if(l_zoom<1.5) l_zoom=1.5;
    Initialize(l_scnSize, l_img_zoom, l_zoom);
}
void Lupa::OnChangeMagnifyOptimize()
{
    UpdateData(TRUE);
    l_optimize=!l_optimize;
}


/*************************************************************************
*
*   Move lupa over the image CDC, responding to (dis)activated status
*
*************************************************************************/
void Lupa::Move(CPoint& a, CPoint& rel, Image* pBmp, CDC* pDC)
{
CPoint p;
CRect r0,r1;
CSize da=a-rel;

if(l_active)    // active lupa was requested
{
    // Remove lupa resizing rectangle, if any
    if(l_dragRect.bottom != 0)
    {
        Draw(pDC);
        l_dragRect=CRect(0,0,0,0);
    }


    if(!l_preactive)    // was not active before
    {
        r0=CRect(0,0,pBmp->m_smS.crScreen.Width(),pBmp->m_smS.crScreen.Height());
        Reset_l_DC(pDC,r0);
        r0=CRect(0,2,0,2);  // dummy update area
    }
    else    r0=l_scnRect;   // Remember previous image area
    // Compute new lupa zoom area
    r1=SetScrnRect(a);
    // For Metheus: ignore rectangles not fully inside the image area
    if(theApp.app_Metheus && !pBmp->m_smS.Screen_in_Image(r1,3))    return;
    l_scnRect=r1;
    // Compute and redraw update rectangles
    bool bu=Update2(r0,r1);
```

```
    if(theApp.app_Metheus)
    {
        r0.InflateRect(2,2);
        pBmp->DisplayDIB(pDC,r0,pBmp->m_smS.Screen_to_Image(r0),CPoint(da));
        /*
        MetheusLoadImageFromDIB(pDC->GetSafeHdc(),l_DIB,theApp.app_DynamicPaletteStart,
                                r0.left,r0.top,r0.Width(),r0.Height(),
                                r0.left,r0.top);
        */
    }
    else
    {
        r0.OffsetRect(-da);
        pDC->BitBlt(r0.left,r0.top,r0.Width(),r0.Height(),l_DC,
                    r0.left,r0.top,SRCCOPY);
    }
    if(bu)
    {
        if(theApp.app_Metheus)
        {
            r1.InflateRect(2,2);
            pBmp->DisplayDIB(pDC,r1,pBmp->m_smS.Screen_to_Image(r1),CPoint(da));
            /*
            MetheusLoadImageFromDIB(pDC->GetSafeHdc(),l_DIB,theApp.app_DynamicPaletteStart,
                                    r1.left,r1.top,r1.Width(),r1.Height(),
                                    r1.left,r1.top);
            */
        }
        else
        {
            r1.OffsetRect(-da);
            pDC->BitBlt(r1.left,r1.top,r1.Width(),r1.Height(),l_DC,
                        r1.left,r1.top,SRCCOPY);
        }
    }
    // Zoom image into new area
    r0=l_scnRect;
    r0.OffsetRect(-da);
    CRect ir=SetImgRect(pBmp->m_smS.Screen_to_Image(a));
    if(l_optimize)
    {
        Image* kadr=new Image();
        if(!kadr->Initialize(8*((7+ir.Width())/8),8*((7+ir.Height())/8),
                             pBmp->m_Bytes_per_Pixel))
        {
            delete kadr;
            pBmp->DisplayDIB(pDC,r0,ir,CPoint(0,0),false);
            pDC->DrawEdge(&(r0),EDGE_BUMP,BF_RECT);
            return;
        }
        kadr->m_pPal->p_active=false;   // no palettes for lupa !
        pBmp->GetSubimage(kadr,ir.left,ir.top);
        kadr->TR_HistStretch(1, false);
        kadr->DisplayDIB(pDC,r0,CRect(0,0,ir.Width()-1,ir.Height()-1),CPoint(0,0),false);
        delete kadr;
    }
    else
    {
        pBmp->DisplayDIB(pDC,r0,ir,CPoint(0,0),false);
    }
    pDC->DrawEdge(&(r0),EDGE_BUMP,BF_RECT);
}
else    // disactivated lupa was requested
{
    if(l_preactive)
    {
        p=l_scnRect.TopLeft()-da;
        if(theApp.app_Metheus)
        {
            pBmp->DisplayDIB(pDC,l_scnRect,pBmp->m_smS.Screen_to_Image(l_scnRect),da);
            /*
            MetheusLoadImageFromDIB(pDC->GetSafeHdc(),l_DIB,theApp.app_DynamicPaletteStart,
                                    p.x,p.y,l_scnRect.Width(),l_scnRect.Height(),
                                    p.x,p.y);
            */
        }
        else
        {
            pDC->BitBlt(p.x,p.y,l_scnRect.Width(),l_scnRect.Height(),l_DC,
                        p.x,p.y,SRCCOPY);
            delete l_DC;
```

```
        }
        l_preactive=false;
        l_DC=0;
    }
}

}

/*****************************************************************
*
*   Represents update region as two rectangles
*   Returns false if only one rectangle "a" must be updated
*   or true if both "a" and "b"
*
*****************************************************************/
bool Lupa::Update2(CRect & a, CRect & b)
{
    if(a.Width()!=b.Width()  ||  a.Height()!=b.Height() ) return false;
    if(a.EqualRect(&b)) // coinciding rectangles
    {
        a=CRect(a.left,a.top,a.left,a.top);
        return false;
    }
    CRect a1; a1.CopyRect(&a);
    CRect b1; b1.CopyRect(&b);
    if(a.left<=b.left && b.left<=a.right)
    {
        if(a.top<=b.top && b.top<=a.bottom)
        {
            a1=CRect(a.left,a.top,b.left,a.bottom);
            b1=CRect(b.left,a.top,a.right,b.top);
        }
        else if(a.top<=b.bottom && b.bottom<=a.bottom)
        {
            a1=CRect(a.left,a.top,b.left,a.bottom);
            b1=CRect(b.left,b.bottom,a.right,a.bottom);
        }
        else return false;
    }
    else if(a.left<=b.right && b.right<=a.right)
    {
        if(a.top<=b.top && b.top<=a.bottom)
        {
            a1=CRect(a.left,a.top,b.right,b.top);
            b1=CRect(b.right,a.top,a.right,a.bottom);
        }
        else if(a.top<=b.bottom && b.bottom<=a.bottom)
        {
            a1=CRect(a.left,b.bottom,b.right,a.bottom);
            b1=CRect(b.right,a.top,a.right,a.bottom);
        }
        else return false;
    }
    else return false;
    a.CopyRect(&a1);
    b.CopyRect(&b1);
    return true;
}

/*****************************************************************
*
*   Copies current screen image into lupa CDC l_DC
*
*****************************************************************/
void Lupa::Reset_l_DC(CDC * pDC, CRect& scrolled_client)
{
    int w=scrolled_client.Width();
    int h=scrolled_client.Height();
    l_preactive=true;
    if(theApp.app_Metheus)
    {
        l_DC=pDC;   return;

        w=2048; h=2560;
        // BYTE buffer
        if(l_Data)  {   delete [] l_Data;   l_Data=NULL; }
        if(l_DIB)
        {
            MetheusDeleteCompatibleBitmap(pDC->GetSafeHdc(),l_DIB);
```

```
            l_DIB=NULL;
        }
        l_DIB=MetheusCreateCompatibleBitmap(pDC->GetSafeHdc(),w,h);

        l_Data=new BYTE[w*h*3];
        HDC hdc=pDC->GetSafeHdc();
        BOOL b1=MetheusGetImageIntoData(hdc,l_DIB,theApp.app_DynamicPaletteStart,
                                        (UCHAR*)l_Data, w, h, 2*w, 16,
                                        0,0,w,h,
                                        0,0);
        if(b1==FALSE)
        {
            Beep(700,150);
            AfxMessageBox("MetheusGetImageIntoData Failed");
        }
        BOOL b2=MetheusLoadImageFromData(hdc,l_DIB,theApp.app_DynamicPaletteStart,
                                        l_Data, w, h, 2*w, 16,
                                        0,0,w,h,
                                        scrolled_client.left, scrolled_client.top);
        if(b2==FALSE)
        {
            Beep(700,250);
            AfxMessageBox("MetheusLoadImageFromData Failed");
        }
    }
    else
    {
        if(l_DC) {  l_DC->DeleteDC(); delete l_DC; l_DC=0; }
        l_DC=new CDC();
        l_DC->CreateCompatibleDC(pDC);
        CBitmap b;
        b.CreateCompatibleBitmap(pDC,w,h);
        l_DC->SelectObject(&b);
        l_DC->BitBlt(0,0,w,h,pDC,scrolled_client.left, scrolled_client.top, SRCCOPY);
        b.DeleteObject();
    }
}

/*********************************************************************
*
*   Interactively resize the Lupa region on the image
*
*********************************************************************/
bool Lupa::Resize(CDC *pDC, CPoint center, CPoint vertex)
{
    Draw(pDC); // remove old rectangle
    CPoint z=vertex-center;
    int a=(abs(z.x))<<1; if(a<32) a=32;
    int b=(abs(z.y))<<1; if(b<32) b=32;
    Initialize(CSize(a,b), l_img_zoom, l_zoom);
    l_dragRect=SetScrnRect(center);
    Draw(pDC); // draw new rectangle
    return true;
}
bool Lupa::Resize(CDC *pDC)
{
    Draw(pDC); // just remove old rectangle
    l_dragRect=CRect(0,0,0,0);
    return true;
}

/*********************************************************************
*
*   Output Lupa parameters into a string (used in status bar)
*
*********************************************************************/
CString Lupa::toString()
{
    CString s;
    s.Format("Screen size %dx%d, zoom=%.2lf",l_scnSize.cx,l_scnSize.cy,l_zoom);
    return s;
}

/*********************************************************************
*
*   Draw Lupa region rectangle
*
*********************************************************************/
void Lupa::Draw(CDC *pDC)
{
    int dmode=SetROP2(pDC->m_hDC, R2_NOT);
```

```
    pDC->MoveTo(l_dragRect.TopLe   .));
    pDC->LineTo(l_dragRect.right,l_dragRect.top);
    pDC->LineTo(l_dragRect.right,l_dragRect.bottom);
    pDC->LineTo(l_dragRect.left,l_dragRect.bottom);
    pDC->LineTo(l_dragRect.TopLeft());
    SetROP2(pDC->m_hDC, dmode);
}
```